Individual Project – Problem Decomposition

When I start to solve the online shop system, I could find the first objects of my program simply analysing the problem and requirements described in the assignment, combined with the actual online shopping system, particularly understanding the domain and looking at common nouns. Thus, I created the following Java classes.

1. *Commodity*
2. *Basket*
3. *Order*

This project is mainly based on the OOP principles, a kind of programming paradigm. In object-oriented programming, a program is considered to be a series of interacting objects. Classes are the blueprint for making objects, these classes represent entities in a right way. When encapsulating these classes, it is important to show the visualisation of the entities clearly. The constructors include their own fields also created to initialising any instance variables. There are also a number of public methods created which can be used externally to read or change these fields. Such as *Getter* and *Setter*. Finally, encapsulating fields and method within this class and controlling access to them and use them through the exposed interface.

Adhering to the programming principles of low coupling and high cohesiveness, I have divided the whole system process into three parts, first the commodity display phase, second the commodity purchase phase and third the order placement phase. The attributes are represented in a particular class, and the methods are strictly related. For example, in the product class, only the properties of the product are included, as well as a set of methods to operate on the product. When I designed these classes, I tried to minimise coupling and to make them as independent as possible so that they would have as little impact on each other as possible. Furthermore, there will inevitably be some inter-calls in other classes. The programming process also adheres to the principle of reuse, where the same methods can be used in the multiple class, which can be more efficient.

The reason why inheritance is not used to represent these relationships is that inheritance would increase coupling and would break class encapsulation. It is preferable to use composition rather than inheritance. In the *basket* class and the *commodity* class should consider using composition, so that the *basket* class can make good use of the details of the commodity, which would eliminate the coupling between the two classes within *operateBasket* class. Another is that the methods in the *commodity* class should be in a separate place, perhaps in a new class. This would increase the class cohesion of the appropriate classes.

In my opinion, Object-oriented programming has absolutely no advantage in terms of efficiency in software execution. Its main purpose is to make it easier for programmers to organize and manage code, to quickly sort out programming ideas and to bring about a revolution in programming thinking.

```java
package com;
/*this is the entity class of order*/
public class Order {
    private int id;
    private String name;
    private String address;
    private String phoneNumber;

    public Order(int id, String name, String address, String phoneNumber) {...}

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }
```

```java
public class Commodity {
    private int id;
    private String name;
    private double price;
    private String description;
    private int stock;

    public Commodity(){}

    public Commodity(int id, String name, double price, String description, int stock){...}

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }
```

```java
package com;
/*this is the entity class of basket */
public class Basket {
    private int id;
    private String name;
    private double price;
    private int number;
    private double total;

    public Basket(){}

    public Basket(int id, String name, double price, int number, double total){...}

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }
```

Individual Project – Method Choice

Method 1

Method *showAllProducts()* includes three parts use to implement different types of functions. One is to show all commodities and details of them, the other is based on the display of a specific commodity and information.

```
/* show all productions */
public void showAllProducts(Map<Integer, Commodity> map){
    System.out.println(" id " + id
            + " name " + name
            + " price " + price
            + " description " + description
            + " stock = " + stock);
    Set set = map.keySet();
    for (Object object: set){
        System.out.println(map.get(object));
    }
    Scanner scanner = new Scanner(System.in);
    System.out.println("please choose your option ");
    System.out.println("1: add commodity to basket" +
            "\n" + "2: return to main page" );
    int option = scanner.nextInt();
    if (option ==1 ){
        operateBasket.add();
    }else{ return; } }
```

In the *showAllProducts()* method, this method takes one parameter, a *HashMap*, which holds all information about all the commodities. The *keyset()* method is used to store all the keys in the map into the *Set* collection. Since *Set* has an iterator, all the keys are retrieved iteratively, and since there are multiple entries in a map collection, A *for-each* loop is used to print out the details of all the items. Where the data stored in *Set* is mapped to Object and finally the value of each object is obtained according to the *get()* method.

```
/*show productions by productions id*/
public void showAllProducts(Integer id){
    if (index.map.containsKey(id)){
        System.out.println(" id " + id
                + " name " + name
                + " price " + price
                + " description " + description
                + " stock = " + stock);
        System.out.println(index.map.get(id));
    }else{
        System.out.println("sorry, cannot find this commodity");
        System.out.println("----------------"); } }
```

The other *showAllProducts()* method takes one parameter, the product id, it entered from the keyboard, which is retrieved by the *search()* method. The *containsKey()* method determines whether a *Map* collection object contains the specified key name. Returns true if the Map collection contains the specified key name, otherwise it returns false. The if else statement is used to determine whether the commodity can be found.

The *try-catch* block should be considered in these two *showAllProducts()* methods. It could catch the error from keyboard or else. I believe that this function is straightforward to understand and maintain. Because I have separated several steps and kept the code neat and simple. Each functional module is commented so that other programmers can understand the code. However, it is important to improve the standardisation of the code so that the methods can be used in a more logical way in relation to each other.

Method 2

Checkout() method was designed to perform the final calculation of the order. This method including the storage and print of user details, the selection of the payment method and print all information of one order.

This method starts by storing the user's details, using an *ArrayList* to store the user's details, receiving the user's information from the keyboard and then calling the add method in the list to store it in the *Order* class.

After the user information has been added, you will be taken to the payment screen. A pre-made map key-value pair contains several payment methods, and the user can enter the corresponding key to retrieve the corresponding value.

Finally, a *for-each* loop is used to print the information about the purchased item. It is more convenient than a normal for loop and easier to find variables within an array or *HashMap*. The *keySet()* method and get value by key are used to print the names of the products stored in the *HashMap* basket according to their keys. After the user has confirmed payment, the shopping basket is emptied. The *map.clear()* method is used here, but before it is emptied, the data stored in the *mapBasket* is transferred to the *mapOrder* so that the order can be viewed later.

In this method I use two kinds of collections to store data, *HashMap* and an *ArrayList*. *HashMap* collections are unordered, have a variable number of stores, can dynamically add data, and store data in the form of key-value pairs, where keys cannot be repeated, and values can be repeated. *ArrayList* is an ordered collection and has subscripts to allow fast random access to elements. So, I used *HashMap* to store various commodities details and use *ArrayList* to store user details.

I have used nested *If* conditional statements to allow the user to choose their next action. There are actually similar nested judgement statements in several of the methods in this project, which is a good way to help the user make the next decision. Although the computer remembers the execution of the outer layer, which may take up more resources, the compiler will optimise it. Another statement that can form a selection structure is the *switch* statement. The *switch* statement can form a multi-branch selection structure, but while the *if* statement is flexible in structure and can handle any number of branches, the *switch* structure is simple and can handle a limited number of problems.

It seems to me that this method is relatively neatly and well written and easy to maintain and modify as it is all separate blocks of code from each other. Some additional comments should be considered in future development so that others can better understand the meaning and purpose of each block and field.

Individual Project – Other Implementation Choice

I chose the basket class to explain the data types because it is the most representative class. The data types used in this class are also all the data types I use throughout the project.

I have used the following data types.

Int is used to store the number of commodities in the shopping basket and the id of commodities in the shopping basket. Although the commodities id already exists in the commodity class, the commodities needs to be given a new id in the basket class. The reason for using *int* is also to better match the encapsulation class Integer for int in the *HashMap* to map the item's id as key to its value. The encapsulation classes for *byte* and *short* are *Byte* and *Short* respectively. Generally speaking, integers use *int* rather than *byte* and *short*, and there is no need for the *long* type which is twice the size of *int*. Also, because all integer arithmetic operations in the java virtual machine (such as adding, dividing etc) only apply to *int* type and *long* type and not to the smaller types. The data types for the number of commodities and the commodity id are *int*.

String is a string type in Java and is also a reference type, not a basic data type. Strings are widely used In Java programming, Strings are objects in Java, and Java provides the String class to create and manipulate strings. The names of commodities are best stored as strings.

Double, the *double* data type is used for the price of commodities and total price in shopping basket. *Double* is used instead of *Float* because it is a double precision floating point number and *Float* is a single precision floating point number, the precision of *Double* is higher to two decimal places and the price displayed can be more accurate.

External libraries usage

I did not use external libraries. I thought about using *Json* to store user information or commodities information, but I didn't think it was necessary. So, I used a more basic *ArrayList* and *HashMap* to store this information. This was used to reflect the learning outcome of collections in the course.

Client Project – Problem Decomposition

PHYT System mainly to solve the communication and teaching between trainers and clients。Therefore, this system needs a communication module, a profile management module and an exercise module. These are three mainly user stories were assigned to me as well. There is actually a registration and login module, but this is done in collaboration with the group and is not covered in this report. Below I will describe the classes and techniques used in practical programming (mainly focus on Comment feature).

The comments, and details can be represented in Java with *Comment* class and *Details* class. The characteristics of *Comments* could be represented by field for: *id* and *content*. Similarly, *Details* could be represented by field for several parameter such as *height*, *weight*, *level* and etc. The creation of these entity classes adheres to the principles of object-oriented programming.

The *CrudRepository* provides the most basic operations for adding, deleting, changing and checking entity classes. It could make the whole development process became more easily and clearly. Also, repository can make communication with database. So, I created the *CommentRepository* to inherit the methods in the *CrudRepository*. These methods can be used in *Controller* files. That makes workflow better.

```
private static final Logger log = LoggerFactory.getLogger(CommentController.class);
@Autowired
private CommentRepository commentRepository;
private JdbcTemplate jdbcTemplate;

@GetMapping(path = "/addComment")
ModelAndView addNewComment(@RequestParam String content,
                           Comment comment) {
    comment.setContent(content);
    commentRepository.save(comment);
    log.info(comment.toString() + "save to the repo");
    return new ModelAndView( viewName: "forward:/html/consoleClient.html");
}
```

Controller file is meant to be a place which is a proxy class between repository (which communicates with database) and system (which provides logic and usage of retrieved data). Inside class body I declare one attribute – *commentRepository* that allows to communicate with *CommentRepository*. It is because of this connection that I can call specific methods in this controller method. The *@RequestParam* annotation is used to request parameters to bind to method parameters of the controller (it is the annotation for receiving common parameters in *SpringMVC*). In this method *org.slf4j.Logger* is used to print and output logs so that processes can be better monitored.

Client Project – Method Choice

Method 1

SendMail method

To enhance the registration experience, an email function has been added to the registration process, which will send an email to the email address entered at the time of registration. The implementation of the email sending function uses multiple threads, the reason for using multiple threads is to improve the user experience. Normally, once a page is white screened for 3s or more, it can be turned off by the user, so we use a sub-thread to do the time-consuming emailing after the user submits the form, while the main thread does its own thing, which improves the user experience and does not require the user to wait for the email to be sent. Multi-threaded processing also can be called asynchronous processing.

```
public class Sendmail implements Runnable {

    private String from = "1044902968@qq.com";
    private String username = "1044902968@qq.com";
    private String password = "ohvfjrjfoyvzbbed";
    private UserDto user;

    public Sendmail(UserDto user){
        this.user = user;
    }
```

The first step is to define the email address that will be used to send the user, in this case QQ email, a popular email address in China. Because the mailbox needs to log into the SMTP server, the username and authorization code should be defined again.

```
    @Override
    public void run(){
        try {
            Properties prop = new Properties();
            prop.setProperty("mail.host","smtp.qq.com");
            prop.setProperty("mail.transport.protocol","smtp");
            prop.setProperty("mail.stmp.auth","true");

            MailSSLSocketFactory sf = new MailSSLSocketFactory();
            sf.setTrustAllHosts(true);
            prop.put("mail.smtp.ssl.enable","true");
            prop.put("mail.smtp.ssl.socketFactory",sf);

            Session session = Session.getDefaultInstance(prop, new Authenticator(){
                public PasswordAuthentication getPasswordAuthentication(){
                    return new PasswordAuthentication(username,password);
                }
            });
            session.setDebug(true);

            Transport ts = session.getTransport();

            ts.connect( host: "smtp.qq.com",username,password);
```

After setting up the mail server, the mail delivery protocol and the need to verify the username and password, you will also need to set up SSL encryption. *MailSSLSocketFactory* is an SSL socket factory that makes it easier to specify trust. The session is created to enable the debug mode of the session so that you can see the status of the emails being sent. Declaring an attribute - *ts* allows me to use the methods in the *Transport* class (a class from *javax.mail* package which is an abstract class that models a message transport). This declaration allows the connection object to the SMTP server to get the transport object to send the mail.

```
    MimeMessage message = new MimeMessage(session);
    message.setFrom(new InternetAddress(username));
    message.setRecipient(Message.RecipientType.TO, new InternetAddress(user.getUsername()));
    message.setSubject("Welcome to team4 PHYT");
    message.setContent(
            o: "<p><h2>Congratulations!</h2></p>"+
            user.getFull_name()+
            "<br>your register email is "+
            user.getUsername()+
            "<br>your user type is "+
            user.getUser_type(),
            type: "text/html;charset=UTF-8");
    ts.sendMessage(message,message.getAllRecipients());
    ts.close();

}catch (MessagingException e){
    e.printStackTrace();
}catch (GeneralSecurityException e){
    e.printStackTrace(); } }}
```

Create a message object with *MimeMessage* (a class represents a MIME style email message) and specify the sender and recipient of the message as well as the message content. Finally close the connection object after sending.

The try-catch block is used here to catch errors and helps with troubleshooting. The same spacing, indentation and the addition of blank lines when moving on to the next functional code block are used in the programming process in the pursuit of code readability. Comments should be added to each functional block to increase code readability and future maintenance and upkeep.

Method 2

UploadMultipleImages

```
@RequestMapping("/multipleImageUpload")
public List multipleImageUpload(@RequestParam("uploadFiles") MultipartFile[] files){
    System.out.println("Number of pictures uploaded:"+files.length);
    List<Map<String,Object>> root=new ArrayList<Map<String,Object>>();
    for (MultipartFile file : files) {
        Map<String,Object> result=new HashMap<String, Object>();
        String result_msg="";
        if (file.getSize() / 1000 > 100){
            result_msg="Picture size cannot exceed 100KB"; }
        else{
            String fileType = file.getContentType();
            if (fileType.equals("image/jpeg") || fileType.equals("image/png") || fileType.equals("image/jpeg")) {
                final String localPath="C:\\Projects\\IdeaProjects\\team4_fitness_and_wellbeing_app" +
                        "\\src\\main\\resources\\static\\ExerciseImg";
                String fileName = file.getOriginalFilename();
                String suffixName = fileName.substring(fileName.lastIndexOf( str: "."));
                //fileName = UUID.randomUUID()+suffixName;
                if (FileUtils.upload(file, localPath, fileName)) {
                    String relativePath= "static/img/" +fileName;
                    result.put("relativePath",relativePath);
                    result_msg="Picture uploaded successfully"; }
                else{ result_msg="Image upload failed"; } }
            else{ result_msg="Incorrect image format"; }
        }
        result.put("result_msg",result_msg);
        root.add(result); }
    String root_json=JSON.toJSONString(root);
    System.out.println(root_json);
    return root; }
```

In the Exercise module, it is requested that some images can be uploaded to be used for client training. The main purpose of this function is to upload images and record information about them. Method requires two arguments, first one is a query parameter “uploadFiles” received from HTML form, and another is declared an attribute “files” (from interface *MultipartFile*, which is implement *InputStreamSource*. A number of methods in this interface can be called to get the main parameters of the file, such as getting the file size, the file type, the original name of the file, etc.)

In this method, a *for-each* loop was be used to loop save files and some *if* conditions were used to determine the size of the file, judging the type of file and getting the file name. Furthermore, *Json* is used to serializes the specified object as the equivalent Json representation and output the information generated during the upload process.

```java
import org.springframework.web.multipart.MultipartFile;
import java.io.File;
import java.io.IOException;
public class FileUtils {
    public static boolean upload(MultipartFile file, String path, String fileName) {
        String realPath = path + "\\" + fileName;
        System.out.println("upload file: " + realPath);
        File dest = new File(realPath);
        if (!dest.getParentFile().exists()) {
            dest.getParentFile().mkdir();
        }
        try {
            file.transferTo(dest);
            return true;
        } catch (IllegalStateException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            return false;
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            return false; } }}
```

It is worth noting that the controller's role in this method is to judgement and pass information. The actual upload operation is done mainly by the *upload()* method in *FileUtils.class*, where the file is saved in the path set by the controller by calling the *transferTo* method in *MultipartFile*. *Try-catch* block was used here to catch *IllegalStateException* and *IOException*. The *if* conditional statement is to determine if the parent folder exists.

This method has been developed in two parts, one for uploading and one for control, which makes it easy to maintain and improves readability, but some comments should be added to these methods. I found that I have a bad programming habit of not commenting often enough and this needs to be corrected in future development.

It is most common for websites to upload images, files etc. directly to a directory on the server, or directly under a specified folder on the server. This approach is really convenient and simple for simple standalone applications, and fewer problems will arise. However, for distributed projects or large-scale project, the direct upload to the project path is obviously unreliable, and as the volume of business increases, so do the files, which naturally puts more pressure on the server. Uploading to a third-party content store could be considered in future improvements.

Client Project – Other implementation choices

In development of *Comment* class and *Details* class I used following data types.

```
import org.hibernate.annotations.GenericGenerator;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.io.Serializable;
@Entity
public class Details implements Serializable {
    @Id @GeneratedValue(generator="system-uuid")
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    private String id;
    private String sex;
    private String height;
    private String weight;
    private String level;
    private String speciality;
    private String qualification;
    private String organisation;
```

@Enity annotations is used in Comment class and Details Class, this annotation is kind of *Java Persistence API*. *@Entity* could annotate the entity. Declare comments and details *id* as the primary by using *@Id* and use *@GeneratedValue* to specify the primary key generation strategy as *IDENTITY*. One of the main goals of *Java Persistence API* is to provide a simpler programming model. It can be easily integrated with other frameworks or containers.

@GeneratorValue belongs to a *JPA* interface which contains two abstract parameters, a strategy of type *@GenerationType* and a generator of type *String*, and both parameters have corresponding default values.

```
@Autowired
private DetailsRepository detailsRepository;
```

Another data type that I used was *DetailsRepository*. I needed attribute of this type in order to access certain method from this class.

About external libraries.

```
import com.alibaba.fastjson.JSON;
```

External library *JSON* was used in *ImageUploadController* class. It can convert Java objects to *JSON* format fast, and of course it can also convert *JSON* strings to Java objects.
https://github.com/alibaba/fastjson

Another external library is *com.sun.mail*

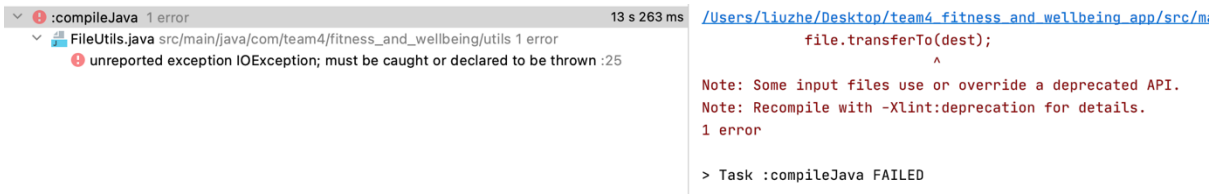
```
import com.sun.mail.util.MailSSLSocketFactory;
```

I am not really sure that this is an external library. But it needs Gradle to compile. This library mainly focuses on mail sending.

Client Project -Error Reporting

Three errors that have been encountered during the process are two compile-time errors and one logic error.

Compile time



This compile error is in the *FileUtils* class because this method needs to throw exceptions (*IllegalStateException* and *IOException*), the solution is to add a *try-catch* block to catch these exceptions or declared to be thrown.

```
public static boolean upload(MultipartFile file, String path, String fileName) throws IOException, IllegalStateException  {
```

or

```
try {

    file.transferTo(dest);
    return true;
} catch (IllegalStateException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
    return false;
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
    return false;
}
```

Run time

```
final String localPath="/static/ExerciseImg";
```

Relative path

```
upload file: /static/ExerciseImg\pic3.jpg
java.io.IOException: java.io.FileNotFoundException:
```

Cannot work with relative path

```
final String localPath="C:\\Projects\\IdeaProjects\\team4_fitness_and_wellbeing_app" +
        "\\src\\main\\resources\\static\\ExerciseImg";
```

Absolute path

This is a runtime error due to a file path problem where it cannot use a relative path but must use an absolute path to save the image, I have tried to fix this so that the relative path also works but after several attempts it has failed. One disadvantage of absolute paths is that they cannot be run on other machines, for example a Macintosh does not even have a c drive. In the future, network storage or third-party storage will need to be used instead of absolute path storage so that programs can be run on any machine.

Run time

```
javax.mail.SendFailedException: Invalid Addresses;
  nested exception is:
    com.sun.mail.smtp.SMTPAddressFailedException: 501 Bad address syntax. http://service.mail.qq.com/cgi-bin/help?subtype=1&&id=20022&&no=1000730

    at com.sun.mail.smtp.SMTPTransport.rcptTo(SMTPTransport.java:2079)
    at com.sun.mail.smtp.SMTPTransport.sendMessage(SMTPTransport.java:1301)
    at com.team4.fitness_and_wellbeing.utils.Sendmail.run(Sendmail.java:53)
    at java.base/java.lang.Thread.run(Thread.java:834)
Caused by: com.sun.mail.smtp.SMTPAddressFailedException: 501 Bad address syntax. http://service.mail.qq.com/cgi-bin/help?subtype=1&&id=20022&&no=1000730

    at com.sun.mail.smtp.SMTPTransport.rcptTo(SMTPTransport.java:1932)
```

```
message.setRecipient(Message.RecipientType.TO, new InternetAddress(user.getUsername()));
```

This is also a runtime error, where the username is entered as a non-email when registering. And *getUsername()* method receives an incorrectly formatted email name in the back-end. The solution should be to add validation to the user registration so that if the user enters a non-email address they are alerted or prevented from proceeding to the next step. Strictly speaking this should be an improvement on the web side.

Client Project – Testing

```
@SpringBootTest
@AutoConfigureMockMvc
@TestPropertySource(locations = "classpath:application.properties")
public class JustJunitTest {

    @Autowired
    private CommentController commentController;
    @Autowired
    CommentRepository commentRepository;
    @Autowired
    DetailsRepository detailsRepository;
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private DetailsService detailsService;
    @BeforeEach
    void testBeforeEach() { System.out.println("Start to test"); }
```

As the image above shows, to configure these classes in the correct way I am using annotations from *Junit* and the *Spring boot* framework, specifying through class annotations that the entire *Spring boot* application context should be set up to run the tests. This can effectively reduce the test run time. Furthermore, I have declared an annotation named *@AutoConfigureMockMVC* to enable and configure auto-configuration of *MockMvc*.

Test 1

```
@Test
public void contextLoads() throws Exception{
    assertThat(commentController).isNotNull();
}
```

The first test class I wrote was a very simple one to check if the *Controller* existed and the test passed with no problem.

Test 2

```
@Test
public void testCommentSave(){
    Comment comment = new Comment();
    comment.setId(1);
    comment.setContent("test");
    commentRepository.save(comment);

}
```

```
@Test
public void testDetailsListAll(){
    Details details = new Details();
    details.setId("1");
    details.setSex("test");
    details.setHeight("test");
    details.setWeight("test");
    details.setLevel("test");
    details.setSpeciality("test");
    details.setQualification("test");
    details.setOrganisation("test");
    detailsRepository.save(details);
    System.out.println(detailsRepository.findAll());
    detailsRepository.findAll();
}
```

The second method I want to test the *repository* class for a series of operations on the database. At first, is a *save()* method for the comment, where the save method of the *commentRepository* is called to save the pre-set values to the comment. The next one is to test the method that lists all the user details. Here the *save()* method in the *detailsRepository* is called to save the pre-set values to the entity class and then the *findAll()* method in the *detailsRepository* is understood to list them. This test also passes nicely.

Test 3

```
@Test
public void testUpdateDetails() throws Exception{
    Details details = new Details();
    when(detailsService.update(details)).thenReturn(true);
    this.mockMvc.perform(post( urlTemplate: "/showDetails"))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(model().hasNoErrors())
            .andExpect(view().name( expectedViewName: "showdetails"));
}
```

First an attribute called *detailsService* is declared as a *@Mockbean*. We use *@MockBean* to create and inject a mock for the *detailsService* and use *Mockito* to set its expected value. The main test in this test is to update the user details in the database, this method should return a *boolean* value “true" if the user details are successfully updated, false otherwise. Following this requirement, I wrote a skeleton of the *update()* method in *DetailsService* class.

To execute a request with *perform()*, which need to pass in a *MockHttpServletRequest* object, followed by *andDo()* to execute a print operation with normal processing. Then comes the execution of the expected match. The *status.isOk()* method is used here, mainly to check whether the expected corresponding success, *hasNoError()* method to verify that the page has no errors, at last *view.name()* method is used to verify the view.

I did not use unit tests during development but only tested afterwards. Unit testing is very important for the quality of a software product, it is the lowest level of all testing, it is a very important part of the whole software testing process, the purpose is to prevent bugs from getting out of hand later in the development process, the later the bugs are found, the more expensive it is to fix them, and the trend is exponentially increasing. I have encountered many bugs during the development process, some are interface problems, some are method problems, if I have timely use of unit testing in the development process, I can precisely find which specific interface or method is buggy.